Prefix Based Numbering Schemes for XML: Techniques, Applications and Performances

Virginie Sans, Dominique Laurent
ETIS – CNRS ENSEA Univ Cergy-Pontoise
F-95000 Cergy Pontoise
+33 1 34 25 66 35
{virginie.sans, dominique.laurent}@u-cergy.fr

ABSTRACT

Commonly used in network routing, programming, classification and knowledge representation systems, labeling schemes have also interested the XML community. We thus motivate and describe numbering schemes, their applications, and the trade off between storage capacities and runtime performance. We present a taxonomy of numbering schemes for XML based on the types of supported queries (ancestor, adjacent, etc), the encoding technique, and whether the scheme offers robustness properties according to updates. We describe some of the numbering techniques proposed for XML. We focus on prefix-based schemes. We give a qualitative comparison of the existing numbering schemes, discussing their advantages and drawbacks. Then, we compare their storage requirement and performances. Finally, we consider the new research directions that are likely to benefit from numbering scheme techniques.

1. INTRODUCTION

Labeling schemes are widely employed for different purposes as network routing [31], programming [30, 32, 33], classification and knowledge representation systems [23, 34], and have regained interest with XML techniques [8, 11, 13].

One of the most interesting properties of XML is its capability of representing hierarchical data, and with this property come two requirements that are decision and reconstruction [14]. The decision requirement aims to decide if two nodes satisfy a binary relation such as ancestor, adjacent, etc, and the reconstruction requirement allows reconstructing a tree starting from a set of nodes. Decision and reconstruction can be achieved more efficiently if the document/tree is associated to a numbering scheme.

Several numbering schemes have been proposed in the past [1-22], and can be divided into two families, namely interval based schemes and prefix based schemes. We recall the basics of these families below.

1.1 Interval Based Schemes

PVLDB '08, August 23-28, 2008, Auckland, New Zealand

The most representative work related to the interval based schemes family is the work by Li and Moon [3]. In this work, the authors propose a technique whereby identifiers are represented as intervals. Using the various related containment predicates, this technique aims to determine if there exists a relationship ascendance/precedence between two given nodes. To that purpose, a pair ($order(x)$, $size(x)$) is associated to each node x in the document in such a way that, for every child node y of x,

$$order(x) < order(y) \text{ and } order(y) + size(y) \leq order(x) + size(x).$$

Therefore, we have the following property

$$[order(y), order(y) + size(y)] \subset [order(x), order(x) + size(x)]$$

if and only if y is the child of x.

When inserting a child to an existing node, it is always possible to find an interval that satisfies the property above. The computation of a new interval for a sibling between two nodes depends on the available remaining space.

A similar approach has been proposed in [19], where sectors are used instead of intervals. The major difficulty of such techniques is to choose the initial size of intervals to minimize the storage cost and to avoid frequent recomputation. The BIRD proposal [14] improves performance with a specific algorithm for choosing the values of $size(x)$. The work of [23] uses float values for defining intervals. Even if the robustness issue is solved, the representation of float values in memory requires integer values. Then, the problem is clearly that the number of integer values between two floats is finite [24, 25], and this entails that interval based schemes robustness depends on the size of intervals and on the integer representation in memory.

1.2 Prefix Based Schemes

Prefix-based schemes directly encode the father of a node in a tree, as a prefix of its label using for instance a depth-first tree traversal. The simplest algorithm is the Dewey Decimal Coding (DDC) widely used by librarians [23]. Dewey encoding is illustrated in Figure 1.(*a*). Let A be a tree with root r and u a node of A. A node n of A is associated with an identifier of the form $key(n).pos(n)$, where $key(n)$ and $pos(n)$ are defined as follows:

- If n is the root r of A then $key(n) = 1$ and $pos(n)$ is undefined.
- Otherwise, assuming that n is the i^{th} child of node v, we have: $pos(n) = i$ and $key(n) = key(v).pos(v)$

In this approach, labels for a tree A can be computed in time linear in the number of nodes in A. Other prefix schemes have been

proposed, namely LSDX, FLEX, ORDPATH, Gabillon *et Al.*, Khaing *et Al.* [16, 26-29]. These schemes are presented in this paper.

A particular sub-class of prefix based scheme is the bit-vector based scheme set, whereby the identifier of a node is seen as a vector of bits. A bit value 1 at a given position identifies a node in a lattice, and each node inherits the bits of its ancestors in a top-down encoding. Several bit vector schemes have been proposed in the literature [32, 33, 34].

1.3 Motivation of the Work

Both interval and bit-vector encoding schemes were initially designed to determine the parent-child relationship between two given nodes. The major advantages of bit-vector schemes are that (*i*) the decision process uses a bit comparison and runs in constant time, and (*ii*) the construction time is linear. However, bit-vector schemes are inappropriate for various XML applications in which updates occur frequently. Interval based schemes are more efficient in the case of XML maintenance, but the major difficulty is, as mentioned above, to choose the initial size of interval, so as to minimize the storage cost and to avoid frequent recomputation.

Many XML applications handle dynamic data that are frequently updated or queried via XPath or XQuery. With the amount of available data, it is not always possible to determine in advance the size of the data and the number of updates. So, both interval and bit-vector schemes are not appropriate for encoding large dynamic XML data. Prefix schemes appear to be more appropriate. They are used for XML maintenance [28, 35], DBMS systems [26, 27, 29], and XML data indexing [16].

Although each of these techniques have been shown appropriate for specific applications, as far as we know, these techniques have not been compared to each other. Only comparisons of interval based techniques versus static prefix based schemes have been studied [10, 5]. Our contribution is to compare existing prefix based works by means of an experimental and analytical evaluation.

The paper is organized as follows. Section 2 presents a brief taxonomy of numbering schemes for XML. In Section 3, we review the main prefix based labeling schemes in terms of our taxonomy. Section 4 describes our experimental evaluation of selected schemes, in terms of their efficiency in decision and reconstruction and of their storage consumption. Finally, in Section 5, we consider future research directions that are likely to benefit from numbering schemes.

2. TAXONOMY

Numbering schemes are classified according to the following four criteria: structure, expressivity, performance and robustness.

2.1 Structure

An XML document is commonly represented as a tree, but new features like XLink and cross-references between documents tend to consider a graph structure instead of a tree structure.

2.2 Expressivity

In [30], numbering schemes are used to decide if a relation of precedence holds between two elements. In this work, the author also studies the order maintenance problem in case of updates in a list (insertion, deletion). In the case of XML, as we consider trees, it not sufficient to consider the order maintenance problem for lists only. Indeed, queries on XML documents are built using path-expressions based on languages like XPath/XQuery or XSLT. It is then necessary to explore the following axes that are possible with existing path-expressions: ancestor, ancestor-or-self, attribute, child, descendant, descendant-or-self, following, following-sibling, namespace, parent, preceding, preceding-sibling, and self.

2.3 Performance

The performances of various encoding and optimization techniques are of course different from each other. In this paper, we compare performances according two criteria, namely the time required for the computation of identifiers of a given XML document, and the space for storing these identifiers.

2.4 Robustness

Renumbering might be necessary in case of updates. Numbering schemes are then divided into two sub-classes: static numbering schemes and dynamic numbering schemes. The robustness of a scheme is the property to support updates without renumbering.

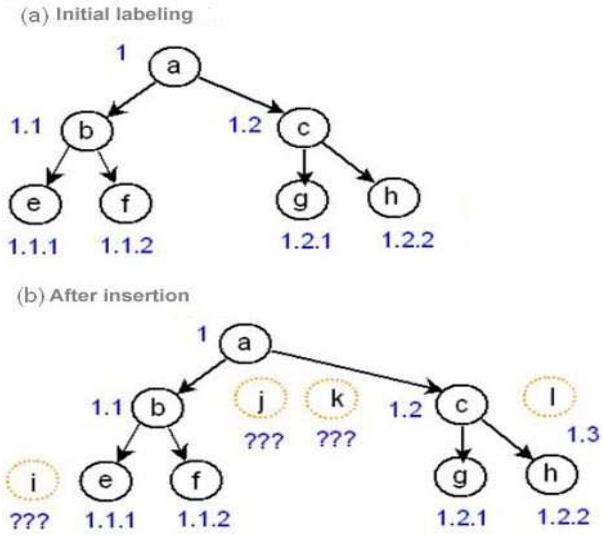

Figure 1. Dewey numbering scheme

In Figure 1(*a*), we give an example of Dewey encoding technique. Assume that a node is to be inserted on the right of node 1.1 or 1.2 (see Figure 1(*b*)). Clearly, no identifier is left for the new node. In this case, the scheme is called *static*, since it requires a complete recomputation of the identifiers. Such schemes, that require recomputation in the presence of updates, are considered as *not robust* [20, 13, 16, 18].

On the other hand, dynamic schemes limit the number of recomputations. They are divided into two families:

- *Range limited robustness* techniques, according to which the frequency of relabeling depends on the number of available identifiers. This family is mainly composed of interval and region based schemes [30, 3].

- *Persistent* techniques according to which relabeling is required only for a very limited subset of update operations. For example, the insertion of a parent node in a prefix scheme [28, 29].

Update operations to be considered on XML documents are the following:

- *Insertion of a node or of a sub-tree*. Insertion can be made on the left/right of a node (for siblings), below a node (child nodes) or above (insertion a new parent for example).
- *Deletion of a node or of a sub-tree*. The position of a deletion is specified as given above in the case of insertion.
- *Modification*. Modification of a node does not affect its position, therefore this case is out of the scope of the paper.
- *Move*. A move operation is managed as a deletion operation followed by an insertion.

Some numbering schemes are affected by *collisions*. A collision happens when the identifier of a newly inserted node already exists in the document. Even though a collision can be predicted, the recomputation of the whole set or a subset of identifiers is required. Moreover, such a prediction can be time consuming.

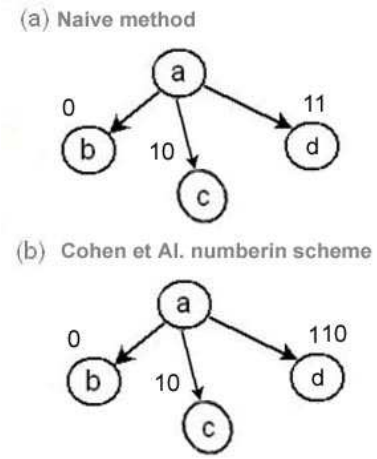

Figure 2. Cohen *et Al.* and naive numbering schemes

In this paper, we study prefix based schemes according to the criteria mentioned above. However prefix based schemes are only dedicated to tree structures. So, the structure dimension is out of the scope of our work. Consequently, we consider the other three criteria, that is expressivity, performance and robustness.

In the next section, we present successively the techniques introduced by Cohen *et Al.*, LSDX, Khaing *et Al.*, Gabillon *et Al.*, and those called Flex Keys and ORDPATH.

3. OVERVIEW OF EXISTING APPROACHES

3.1 Cohen *et Al.* [5]

In [5], a naive prefix numbering scheme based on bits is considered. Each node n is assigned a binary string, denoted by $key(n)$, which represents its position among its siblings. Then, given a path from the root to any node of a tree, each node is assigned the concatenation of all identifiers of the successive nodes encountered. Figure 2(*a*) illustrates this approach in the case of a tree composed of four nodes: the root and its three children.

Of course, if the construction of the keys of sibling nodes is achieved using consecutive bit strings, renumbering is necessary when inserting a new sibling node (except when the new node is to be inserted on the right of the last sibling). To cope with this problem, Cohen's scheme reserves some identifiers for later insertions. For example, in Figure 2(*b*), the children of the root node are labeled with 0, 10 and 110. In this case, the insertion of a new node *e* between nodes *c* and *d* node can be achieved without any renumbering, since *e* can be assigned the key 1110.

More generally, the key of the i^{th} node among its siblings is built up with $(i - 1)$ bits equal to 1 followed by 0.

Moreover, in this case, it can be seen that a bit to bit comparison allows determining the sibling and the precedence relationships.

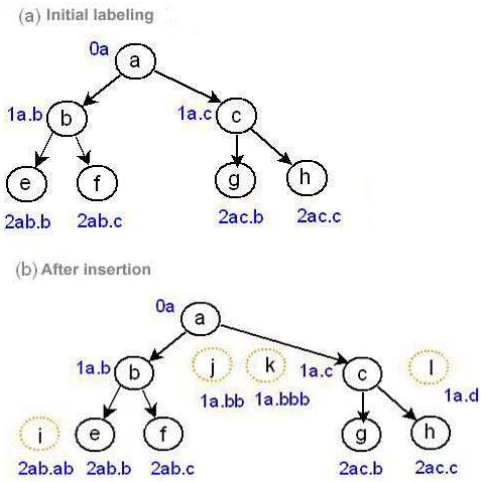

Figure 3. LSDX numbering scheme

3.2 LSDX [16]

LSDX numbering scheme has been proposed by Duong *et Al.* [16]. The LSDX identifier of a node n denoted by $Lsdx(n)$, is of the form $prefix(n).str(n)$, where:

- $prefix(n)$ is the concatenation of $level(n)$ and $code(n)$. $level(n)$ is the depth of n, and of $code(n)$ concatenates $code(v)$ and $str(v)$ where v is the parent node of n,
- $str(n)$ is a string that identifies n among its siblings.

An example of LSDX labeling is given in Figure 3. More generally, let A be a tree whose root is denoted by r. We assume that $level(r) = 0$ and $str(r)$ is the character a. Then, $Lsdx(r) = 0a$, because $code(r)$ does not exist as r has no parent. For every child v_i of r, $level(v_i) = level(r) + 1 = 1$ and $str(v_i)$ is a character which enables to identify the position of this child among its siblings

When a node w is inserted as a sibling of u, the authors give the following rules for the computation of $Lsdx(w)$:

- *level*(*w*) = *level*(*u*)
- *code*(*w*) = *code*(*u*)
- If *w* is inserted on the left of node *u*, and if *u* has no left sibling, then *str*(*w*) = a*str*(*u*). Figure 3(*b*) shows an example of this case where *u* is the node *e*, in which case *Lsdx*(*w*) = 2ab.ab.
- If *w* is inserted between node *u* and node *v*, then *str*(*w*) is chosen so as *str*(*u*) < *str*(*w*) < *str*(*v*),. (see Figure3(*b*) and the nodes identified by 1a.bb and 1a.bbb). If *str*(*u*) ends with z, then *str*(*w*) = *str*(*u*)b.

Although this technique allows identifying ascendant/descendant relationships, collisions may occur.

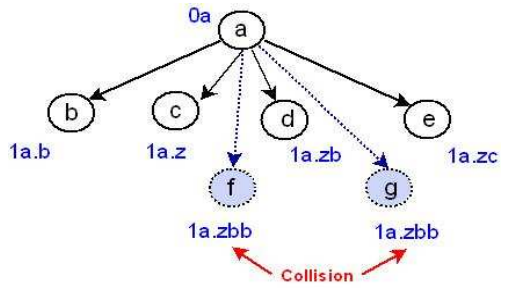

Figure 4. LSDX Collision

For example, referring to Figure 4, the root node has 4 children *b, c, d, e* respectively identified by 1a.b, 1a.z, 1a.zb, 1a.zc. Now, assume that we insert a node between *c* and *d*, and another one between *g* and *e*. According to the given rules, both nodes should be assigned the same identifier, namely 1a.zbb . To prevent collisions, renumbering is possible but the authors have not proposed any clue on when such renumbering is necessary.

3.3 Khaing *et Al.* [26]

Mixing numbers and characters to create identifiers has also been proposed by Khaing *et Al*. [26]. If *n* is a child of *v*, the identifier associated to *n*, denoted by *ID*(*n*), is of the form *prefix*(*n*).*code*(*n*), where:

- *code*(*n*) is the concatenation of a character, denoted by *car*(*n*), and a number denoted by *number*(*n*)
- *prefix*(*n*) is the concatenation of the level of *n* in *A* and *code*(*v*)

The code and the level of the root *r* of *A* are *code*(*r*) = a1 and *level*(*r*) = 0, respectively. As shown in Figure 5(*a*), if u_1 and u_2 are respectively the first and second child of *r*, we have: *ID*(*r*) = 0a1, *ID*(u_1) = 1a1.a1 and *ID*(u_2) = 1a1.b1.

When a node *u* is inserted as a sibling of *v* and a child of *w*, the authors propose the following rules for the computation of *ID*(*u*):

- If *u* is inserted on the left of *v*, and if *v* has no left sibling, then *ID*(*u*) = *prefix*(*u*).*code*(*u*) is such that *prefix*(*u*) = *prefix*(*v*) and *code*(*u*) is the concatenation of *car*(*v*) and (*number*(*v*) - 1). Figure 5(*b*) shows this case for node *i* associated with 2a1a1.a0.
- If *u* is inserted on the right of *v*, and if *v* has no right sibling, then *ID*(*u*) = *prefix*(*u*).*code*(*u*) is such that *prefix*(*u*) = *prefix*(*v*) and *code*(*u*) is the concatenation of *car*(*v*) and (*number*(*v*) + 1). Figure 5(*b*) shows this case for node *l* associated with 1a1.b2.

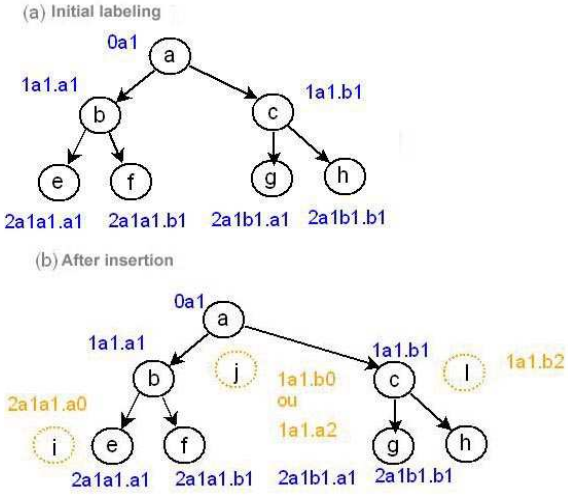

Figure 5. Khaing *et Al.* Numbering Scheme

If *u* is inserted between two sibling nodes *v* and v_2, one of the previous two rules is applied, depending on the implementation. In our example (see Figure 5), node *j* can be associated with identifier 1a1.b0 or 1a1.a2, depending on the implementation.

We note that collisions can also occur in this approach. For example, assume that the first rule above is applied. Then, inserting successively two nodes v_1 and v_2 on the left of a node identified by 1a1.a1, associates them with 1a1.a0 and 1a1.a-1, respectively. Then, assuming that a third node v_3 is inserted between v_1 and v_2, the first rule states v_3 is assigned the identifier 1a1.a-1, which is a case of collision.

3.4 Gabillon *et Al.* [27]

In Gabillon et *Al.* [27], the root of the tree is assigned the identifier (0, 1, 1), and the other nodes are associated with triples defined as follows: A node *u*, child of *v*, is identified by the triple (*level*(*u*), *code*(*v*), *code*(*u*)) such that:

- *level*(*u*) is the level of *u,*
- *code*(*u*) is a pair of integers (n_u, d_u)
- *code*(*v*) is the code of *v*, *i.e.,* a pair (n_v, d_v)

If *u* is a node such that *code*(*u*) = (*i*, 1), then *u* is the i^{th} node at level 1. An example of this technique is given in Figure 6.

In [27], the authors give the following rules for the insertion of a node *w* (see Figure 6(*b*)):

- If *w* is the first node at level 1, then *code*(*w*) = (1, 1)
- If *w* is inserted on the left of *u* with *code*(*u*) = (*i*, *j*) and if *u* has no left sibling, then *code*(*w*) = (*i* - *j*, *j*),
- If *w* is inserted on the right of *u* with *code*(*u*) = (*i*, *j*) and if *u* has no left sibling, then *code*(*w*) = (*i* + *j*, *j*)
- If *w* is inserted between *u* and *v* such that *code*(*u*) = (*i*, *j*) and *code*(*v*) = (*k*, *h*), then *code*(*w*) = (*a*, *b*) where *a* = (*i.h* + *k.j*)/*d*, *b* = (2.*h.j*)/*d*, and *d* = *GCD*((*i.h* + *k.j*), (2.*h.j*)).

Although this technique never requires renumbering, large memory storage is needed for its implementation. We shall come back to this issue in the evaluation section.

3.5 FLEX Keys [28]

FLEX [28] is the acronym of *F*ast *Lex*icographical *Keys*. As shown in Figure 7(*a*), the Flex identifier of a node *n*, denoted by

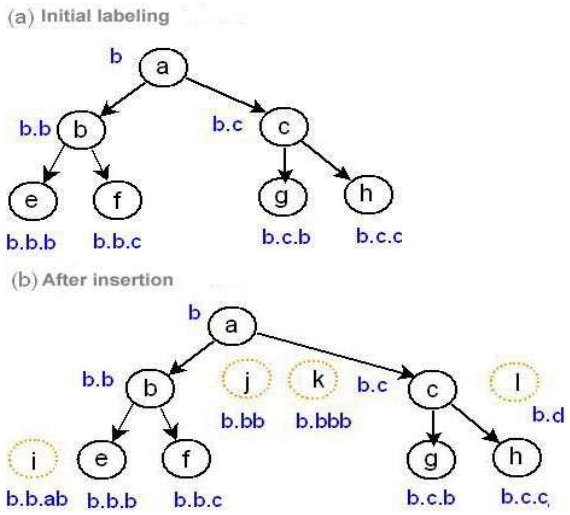

Figure 7. Flex Keys

Flex(*n*), is simply `b` if *n* is the root of the tree, and otherwise, *Flex*(*n*) is of the form *prefix*(*n*).*str*(*n*), where:

- *prefix*(*n*) = *Flex*(*v*), where *v* is the parent node of *n*,
- *str*(*n*) identifies *n* among its siblings *n*.

During the first labeling, the string part of the first child is the character `b` in order to allow future insertions on the left. The rules for creating the string part of Flex identifiers are the same as those

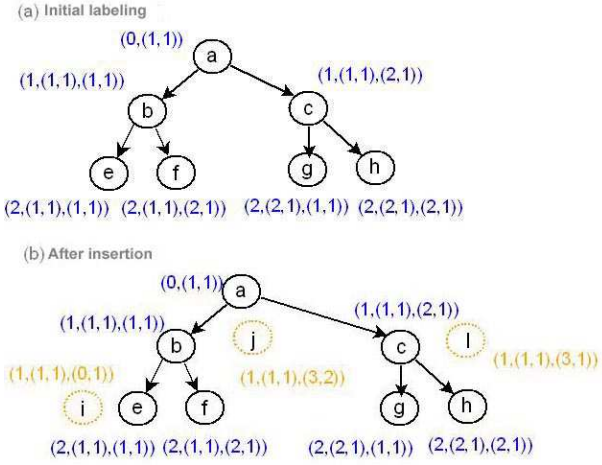

Figure 6. Gabillon *et Al.* numbering scheme

for the *str* component in LSDX [16]. Denoting by *Flex*(*u*) the flex identifier of a node *u*, the properties of the Flex identifiers are:

- If *prefix*(*u*) is a substring of *prefix*(*v*), then *u* is an ancestor of *v*.
- If *prefix*(*u*) = *prefix*(*v*), then *u* and *v* are sibling.
- More generally, if *Flex*(*u*) < *Flex*(*v*) according to the lexicographical ordering, then *u* appears before *v* in the document.

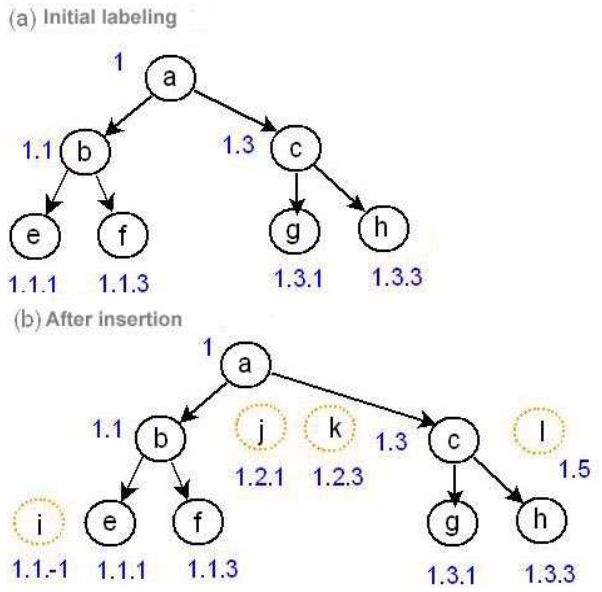

Figure 8. ORDPATH Numbering Scheme

Robustness is the major property of this technique, because renumbering is required in limited cases. Indeed, renumbering is only required when a node is inserted as a parent, in which case all child nodes identifiers of the new node must be recomputed. To the best of our knowledge, no collision problem has been reported, when using this technique.

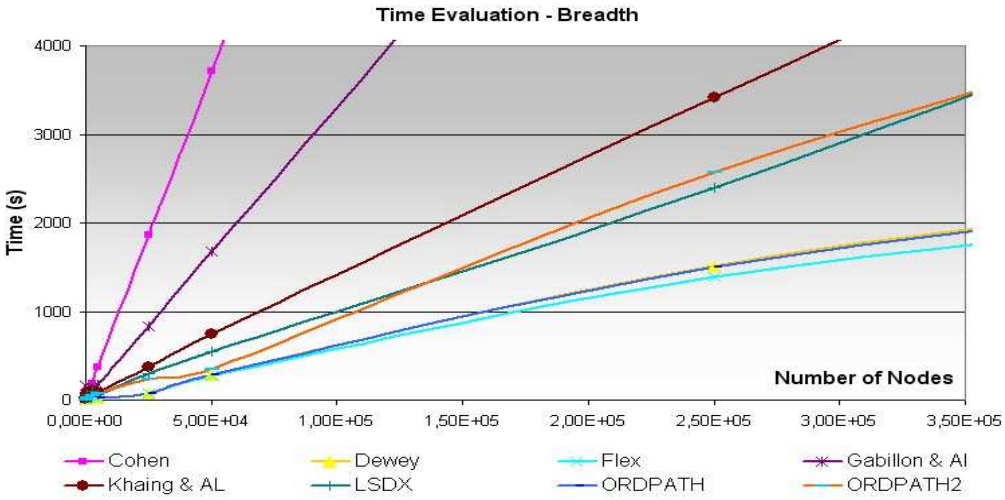

Figure 9. Time evaluation / breadth

3.6 ORDPATH [29]

ORDPATH [29] is a robust technique implemented in commercial applications like Microsoft SQL Server for execution plan or indexation optimization. As shown in Figure 8(*a*), the ORDPATH identifier of a node *n*, denoted by *Ordpath*(*n*), is simply 1 if *n* is the root of the tree, and otherwise, *Ordpath*(*n*) is of the form *prefix*(*n*).*ord*(*n*), where:

- *prefix*(*n*) = *Ordpath*(*v*), where *v* is the parent node of *n*,
- *ord*(*n*) is an integer that identifies *n* among its siblings.

During the first labeling, only odd positive numbers are used as values for *ord*(*n*).

This technique is particularly efficient to encode the parent-child relationship and the sibling relationship. For example, 1.3.1 is a child of 1.3, and 1.3.1 and 1.3.5 are siblings. Moreover, 1.3.1 is a left sibling of 1.3.5.

If *u* is inserted as a sibling of *v*, the authors propose the following rules for the value of *ord*(*u*):

- *ord*(*u*) = (*ord*(*v*) + 2), if *u* is inserted on the right of *v* and if *v* has no right sibling. (see Figure 8(*b*) and the node labeled 1.7).
- *ord*(*u*) = (*ord*(*u*) - 2), if *u* is inserted on the left of *v* and if *v* has no left sibling. (see Figure 8(*b*) and the node labeled 1.-1).
- *ord*(*u*) = ((*ord*(*u*) + *ord*(*u'*))/2).1 if *v* is inserted between the two nodes *v* and *v'*. Notice that, since *ord*(*u*) and *ord*(*u'*) are odd, ((*ord*(*u*) + *ord*(*u'*))/2) is even. (see Figure 8(*b*), and the node labeled 1.2.1).

We note that, given a node *n*, the previous rules imply that the number of dots in *Ordpath*(*n*) followed by an odd number is the level of *n* in the tree.

In our example, identifiers are represented in string formats. However, the authors propose an optimized encoding technique, according to which all numbers appearing in the ORDPATH identifiers are represented by bit strings of a fixed length. We refer to the original paper [29] for more details.

As for Flex Keys, ORDPATH requires recomputation only for parent node insertion.

4. EVALUATION

We have implemented the various numbering schemes recalled previously in Java 1.4.2 and we have used the Sun Microsystems parser, SAX. We have created our own set of XML files, the DTD of which is inspired by bib.dtd of the XQuery Uses cases, where the XML document is a collection of authors elements. We built our own dataset instead of using benchmark tool such as XMark, in order to run tests with different but known values for the breadth and the depth of the documents.

We note that, as the performances of ORDPATH depend on the implementation of the optimized encoding, we have evaluated both possibilities. On the charts, ORDPATH and ORDPATH2 stand respectively for the non optimized and the optimized version. Moreover, we present Dewey encoding as a reference, because it is the most commonly known technique.

Our experiments were performed on a Pentium IV 1.3G with 1024MB RAM, and running the Windows XP OS. We have considered the impact of the depth and breadth of the XML document on the time for generating labels and on the space taken by these labels.

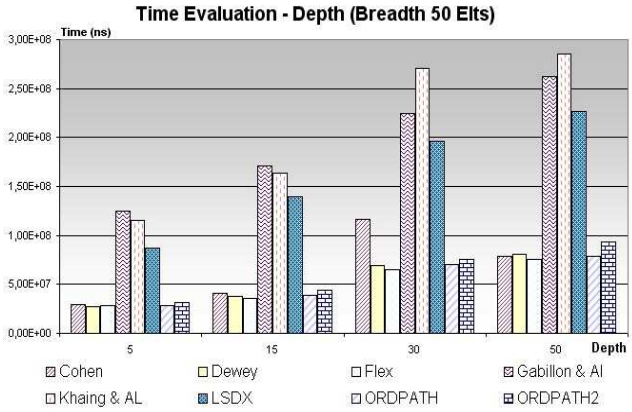

Figure 10. Time evaluation / Depth – constant breadth

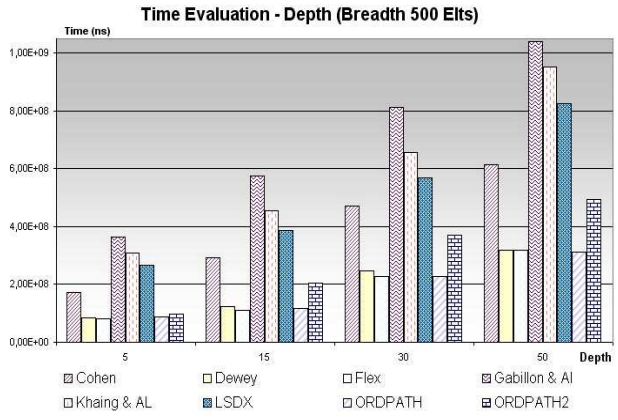

Figure 11. Time evaluation / Depth – constant breadth

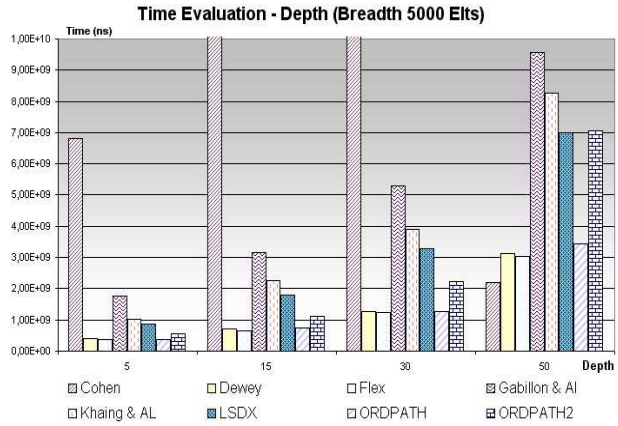

Figure 12. Time evaluation / Depth – constant breadth

4.1 Time Analysis

4.1.1 Breadth Influence

Our first experiment evaluates the required time to generate labels for various datasets that were generated as described above. First, we examine what we call monotonic depth documents. These documents are collections that have the same depth, but in which the number of elements in the collection ranges from 1 to 50,000. Results of this experiment are displayed in Figure 9.

We can see that Flex is the least time consuming technique. Dewey and ORDPATH are not that far from Flex with a computation time below 2,000 s (Breadth = 3.5×10^5 nodes); the worst case is Gabillon *et Al* scheme.

4.1.2 Depth Influence

Our second evaluation considers deep documents. The total depth of the tree ranges from 5 to 500, and for each depth value, the tree may contain 50 up to 5,000 nodes. We also experimented up to 30,000 nodes, but we obtained the same conclusion. Our results are represented in Figures 10-12.

Dewey, Flex and ORDPATH are clearly the best techniques. However, comparing the overall performances of Flex and ORDPATH, Flex appears to be a bit more efficient among dynamic techniques. However, the gain in time is not significant and, as will be seen next, Flex requires more storage space than ORDPATH. Consequently, we consider that ORDPATH is preferable to Flex.

We note that, comparing according to the depth of documents, Gabillon *et Al.* technique does not show good time performance for any type of files. It should also be emphasized that Khaing *et Al.* and LSDX are more efficient for deep documents.

4.2 Storage Analysis

In this section, we are interested in the storage requirement of the numbering techniques. Storage is important for some applications that have little capacities, such as embedded systems, or mobiles and PDAs. To quantify how much the breadth affects the storage, we evaluated the techniques using the same documents as in previous section.

4.2.1 Breadth Influence

Figure 13 displays the size of the identifiers versus the breadth of the document.

Flex uses string whereas ORDPATH (optimized or not) uses bits or numbers, which is more efficient regarding the storage requirement. It should be stressed that with optimization possibilities, ORDPATH is generally the best technique for the storage of identifiers. On the other hand, the approach of Gabillon *et Al.* is inappropriate in terms of space consumption for small files, but shows better storage performance for large documents.

4.2.2 Depth Influence

The depth of the document can have a significant impact on the size of the identifiers. Figures 14-16 list the experiments we run on the various techniques and documents.

Even if Flex is the quickest technique compared to ORDPATH, the exact performance for storage differences vary dramatically with the depth of the document. We observe the same trend on the depth and time as on the depth and space for Gabillon *et Al*. Due to the length of strings used in LSDX proposal, the experiments show that this technique requires a lot of space for storing identifiers when the document breadth increases. We also notice that Cohen numbering scheme requires little storage space, but the time needed to compute the identifiers is generally significant (see Figure 9).

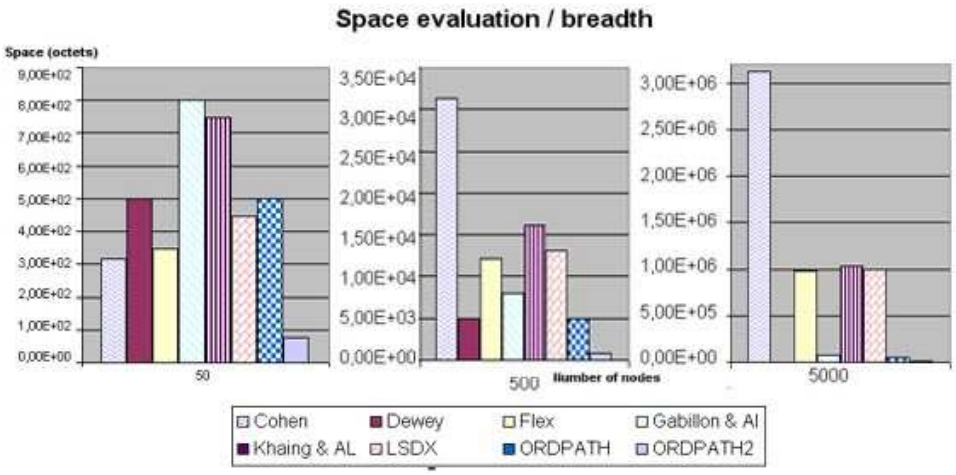

Figure 13. Space evaluation / depth

4.3 Synthesis

In Table 1, we summarize the qualitative and quantitative performances of presented algorithms :

- the time and space performance
- the ratio between time and space performance
- the relationship supported for a decision process
- the existence of collision and renumbering

In Table 1, we assess the numbering techniques according to the considered criteria using marks ranging from A (best) down to E (worst). The main observation on the ratio time/space is that the Flex and ORDPATH time requirements are comparable.

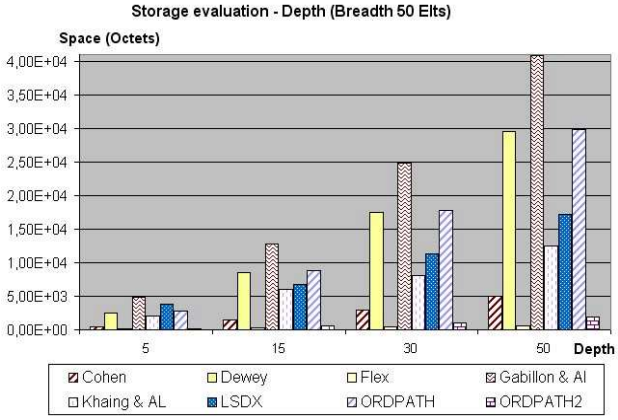

Figure 14. Space evaluation / depth

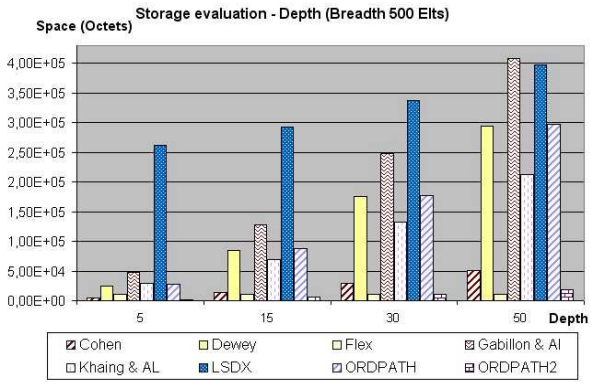

Figure 15. Space evaluation / breadth

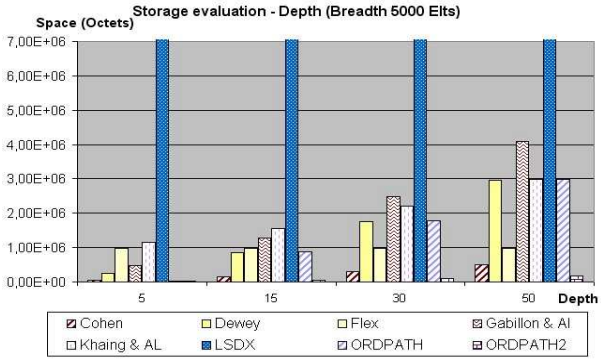

Figure 16. Space evaluation / breadth

We end this section by noting that, in case of collision, as renumbering is necessary, the performances of the different approaches are directly related to the evaluation presented previously. This explains why no experiments have been performed in this respect.

5. CONCLUSION AND FUTURE WORK

In this paper, we have provided a taxonomy on numbering schemes for XML documents. We reviewed various numbering schemes known from the literature and classified them in terms of our taxonomy. We present the results of extensive evaluation experiments, for which these various schemes have been implemented and applied to multi-scale XML documents.

With the integration of heterogeneous contents, many challenges arise. For example, a node can be referenced in several documents (XREF, XLINK), in such a way that each document can be seen as a sub-graph of a XML collection. Specific numbering schemes for graph structure need to be explored. Managing tree and graph structures might be a real challenge for numbering schemes. It is also important to develop comprehensive optimization techniques for existing numbering schemes, in order to fit small or mobile systems (PDA, mobile phone). We note in this respect that, except ORDPATH, no other technique offers optimization possibilities.

Acknowledgements

The authors would like to thank Feroz Mohamad for his help during the experiments.

Table 1. Summary of prefix based techniques

	DEWEY		Cohen		LSDX		Khaing *et Al.*		Gabillon *et Al.*		FLEX		ORDPATH	
	Depth	Breadth	Depth	Breadth	Depth	Breadth	Depth	Breadth	Depth	Breadth	Depth	Breadth	Depth	Breadth
Time consumption	A	B	C	D	E	C	D	E	D	D	A	A	A	B
Storage requirement	B	C	A	D	D	C	B	D	D	D	B	B	B	A
Average ratio Time/Storage	A	B	C	D	E	C	E	D	D	D	A	A	A	A
Persistent	Static		Static		Dynamic		Dynamic		Dynamic		Dynamic		Dynamic	
Robustness	D		D		C		C		A		A		A	
Limitations	mandatory relabelling in case of insertion		mandatory relabelling in case of insertion		mandatory relabelling in case of insertion of parent node		mandatory relabelling in case of Collisions (insertions of siblings)		mandatory relabelling in case of Collisions (insertions of siblings)		mandatory relabelling in case of insertion of parent node		mandatory relabelling in case of insertion of parent node	
decision	self, descendant-self depend on updates		All		following-sibling, preceding-sibling, following, preceding depend on updates		All		All		All		All	

REFERENCES

[1] Santoro N., Khatib R., Labeling and implicit routing in networks. *The Computer Journal*, 28 (1985), 5-8.

[2] Agrawal, R., Borgida, A., Jagadish, H. V., Efficient Management of Transitive Relationships in Large Data and Knowledge Bases, In *Proc. SIGMOD Conf.,* 1989, 253-262.

[3] Li, Q., Moon, B., Indexing and Querying XML Data for regular Path expressions, In *Proc. VLDB Conf.,* 2001, 361-370.

[4] Grustr, T., Accelerating XPath Location Steps, In *Proc. SIGMOD Conf.,* 2002, 109-120.

[5] Cohen, E., Kaplan, H., Milo, T., Labeling Dynamic XML, In *Proc. PODS Conf.,* 2002, 271-281.

[6] Kha, D. D., Yoshikawa, M., Uemura, S., A Structural Numbering Scheme for XML Data, In *Proc. Worshops Xmldm, Mdde, and YRWS on Xml-Based Data Management and Multimedia Engineering-Revised Papers,* 2002, XX-YY..

[7] Takharu, E., Toshiyuki, A., Yoshikawa O., Shunsuke U., A robust XML Node Numbering Scheme and its Management, *IEIC Technical Report* (Institute of Electronics, Information and Communication Engineers) Vol 102 n°208, (2002), 85-90.

[8] Wang H., Park S., Fan W., Yu., P., ViST: A dynamic index method for querying XML data by tree structures, In *Proc. SIGMOD Conf.,* 2003, 110-121.

[9] Wu, X., Lee, M.L, Hsu, W., A prime number labeling scheme for dynamic ordered XML Trees, In *Proc. ICDE Conf.,* 2004, 66-78.

[10] Yu, X.J., Luo, D., Meng, X., Lu, H., Dynamically Updating XML Data : Numbering Scheme revisited, In *WWW: Internet and Web Information System*, 7, 2004, 5-26.

[11] Böhme, T., Rahm, E., Supporting efficient streaming and insertion of XML data in RDBMS, In *Proc. Int. Workshop Data Integration over the Web (DIWeb),* 2004, 70-81.

[12] Yi, C., Mihaila, G., Bordawekar, R., Padmanabhan, S., L-Tree: A Dynamic Labeling Structure for Ordered XML Data, In *Proc. Current Trends in Database Technology - EDBT Workshops*, LNCS 3268, 2005, 209-218.

[13] Kha, D., Yoshikawa, M., Uemura, S., A Structural Numbering Scheme for processing Queries by Structure and Keyword on XML Data, *IEICE Transactions on Information Systems,* 2004, 361-372.

[14] Weigel, F., Schulz, K.U., Meuss, H., The BIRD Numbering Scheme for XML and Tree Databases, Deciding and Reconstructing Tree Relations using efficient Arithmetic Operations , *Database and XML Technologies,* LNCS 3671, 2005, 49-67.

[15] Peleg, D., Informative labeling schemes for graphs, *Theoretical Computer Science,* 340(3), 2005, 577-593.

[16] Duong, M., Zhang, Y., LSDX : A New Labeling Schema for Dynamically Updating XML Data, *In Proc. ADC Conf.,* 2005, 185-193.

[17] Silberstein, A., He, H., Yi, K., Yang, J., BOXes: Efficient Maintenance of Order-Based Labeling for Dynamic XML Data, *In Proc. ICDE Conf.,* 2005, 285-296.

[18] Abiteboul, S., Alstrup, S. ,Kaplan, H., Milo, T., Rauhe ,T., Compact Labeling Scheme for Ancestor Queries, *SIAM J. Comput.,* 2006, 547-556.

[19] Thonangi, R., A concise labeling scheme for XML data, *In Proc. COMAD Conf.*, 2006, XX-YY.

[20] Bremer, J.M, Gertz, M., An efficient XML Node Identification and Indexing Scheme, *Technical Report CSE-2003-04, University of California at Davis,* 2003.

[21] Li, H., Li, L., A DTD-Conscious Sparse Numbering Scheme, In *Proc. of Computer and Information Technology Conf. (CIT)*, 2004, 295-302.

[22] Fisher, D.K, Lam, F., Shui, W.M., Wong, R.K., Dynamic Labeling schemes for ordered XML Based on Type Information, In *Proc. ADC Conf.,* 2006, 59-68.

[23] Tatarinov, I., Viglas, S., Beyer, K., Shanmugasundaram, J., Shekita, E., Zhang, C., Storing and Querying Ordered XML Using a relational database System, In *Proc. SIGMOD Conf.,* 2002, 204-215.

[24] Amagasa, T., Yoshikawa, M., Uemura, S., QRS: A Robust Numbering Scheme for XML Documents, In *Proc. ICDE Conf.,* 2003, 705-707.

[25] Ren, J., Yin, X., Guo, X., A Dynamic Labeling Scheme for XML Document, In *Journal of Communication and Computer*, 3(5), 2006, 61-65.

[26] Khaing, A., Thein, N. L., A Persistent Labeling Scheme for Dynamic Ordered XML Trees, In *Proc. Conf. on Web Intelligence,* 2006, 498-501.

[27] Gabillon, A., Fansi, M., A persistent labelling scheme for XML and tree databases, In *Proc. SITIS Conf.,* 2005, 110-115.

[28] Deschler, K., Rundensteiner, E., MASS: A Multi-axis Storage structure for Large XML Documents, In *Proc. Conf. on Information and Knowledge Management*, 2003, 520-523.

[29] O'Neil, P., O'Neil, E., Pal, S., Cseri, I., Schaller, G., Westbury, N., ORDPATHs: Insert-Friendly XML Node Labels., In *Proc. SIGMOD Conf.,* 2004, 903 - 908.

[30] Dietz, P. F., Maintaining order in a linked list. In *Proc. STOC Conf.,* 1982, 122-127.

[31] Gavoille C., Peleg, D, Compact and localized distributed data structures. *Journal of Distributed Computing, Special Issue for the Twenty Years of Distributed Computing Research*, 2003, 111-120.

[32] Wirth, N., Type extensions. *ACM Trans. on Programming Languages and Systems*, 10(2), 1988, 204-214.

[33] Krall, A., Vitek, J., and Horspool., N., Near optimal hierarchical encoding of types. In *11th European Conf. on Object Oriented Programming (ECOOP'97)*, 1997, 128-145.

[34] Aït-Kaci, H., Boyer, R., Lincoln, P., and Nasr, R. Efficient implementation of lattice operations. *ACM Trans. on Programming Languages and Systems*, 11(1), 1989, 115-146.

[35] Dang-Ngoc., T.T., Sans, V., Laurent, D., Classifying XML Materialized views for their maintenance on distributed Web sources. In *Proc EGC Conf.,* RNTI 2005, 433-444.